Interaktive Computergrafik

Vorlesung im Sommersemester 2014
Kapitel 3: Deferred Shading und Bildraum-Techniken

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

Inhalt

- in diesem Kapitel befassen wir uns mit Methoden, die zum Einsatz kommen wenn **Shading teuer** ist, z.B. aufgrund von
  - Beleuchtungsberechnung durch viele Lichtquellen
    (wir ignorieren zunächst das Problem, dass wir dann auch entsprechend viele Shadow Maps/Volumes benötigen würden)
  - prozeduraler Texturierung
  - teuren Datenstrukturen (3D Texturierung mit Octrees)
  - wenn wir Ambient Occlusion, Tiefenunschärfe, indirekte Beleuchtung usw. (approximativ, d.h. im Bildraum) berechnen möchten
- wir beschäftigen uns in diesem Kapitel noch nicht mit
  - Entfernen von Geometrie außerhalb des View Frustum oder
  - Anpassung der Tessellierung
  - beides ist natürlich ebenfalls wichtig für effizientes Rendering

Motivation

Single-Pass Lighting (für mehrere Lichtquellen)

- einfachste Möglichkeit: ein Vertex- und Fragment Programm für alle Aufgaben

```
für jedes Objekt
  zeichne Objekt, behandle alle LQ in einem Shader
```

- Problem: auch verdeckte Flächen werden mit einem (teuren) Fragment-Programm rasterisiert

- Handhabung verschiedener Materialien und Lichtquellen schwierig
  - Verzweigungen für Material-Lichtquellen-Kombinationen
  - begrenzte Anzahl von Interpolatoren (GLSL in/out), d.h. nicht beliebig viele Werte pro Vertex deren Resultat weitergegeben werden kann

- Schatten
  - Stencil Schattenvolumen sind so nicht einsetzbar
  - Shadow Mapping: große Anzahl von Shadow Maps stellt kein Problem dar

Motivation

Multi-Pass Lighting

- Rendering mit Trennung nach Lichtquellen

```
für jede Lichtquelle
  für jedes Objekt (optional: im Einflussbereich der LQ)
    framebuffer += Beleuchtung( Objekt, Licht )
```

- einfachere Shader
- aber viele Rendering-Aufrufe: Anzahl Objekte × Anzahl Lichtquellen
- nach wie vor: unnötiger Aufwand für verdeckte Flächen

- Hauptproblem: viele Aufgaben fallen mehrfach an
  - Geometrieverarbeitung
  - Rasterisierung
  - Texturfilterung
  - ...

Motivation

Depth-Only Pass

- GPUs unterstützen einen speziellen „Depth-Only“-Render-Modus
  - fülle in ersten Render-Durchgang nur den Tiefenpuffer:
    Rendering schneller, wenn Schreiben in den Frame-Buffer deaktiviert
  - zeichne dann das eigentliche Bild, wobei die GPU dafür sorgt, das für verdeckte Flächen (fast) kein Fragment-Shader angestoßen wird

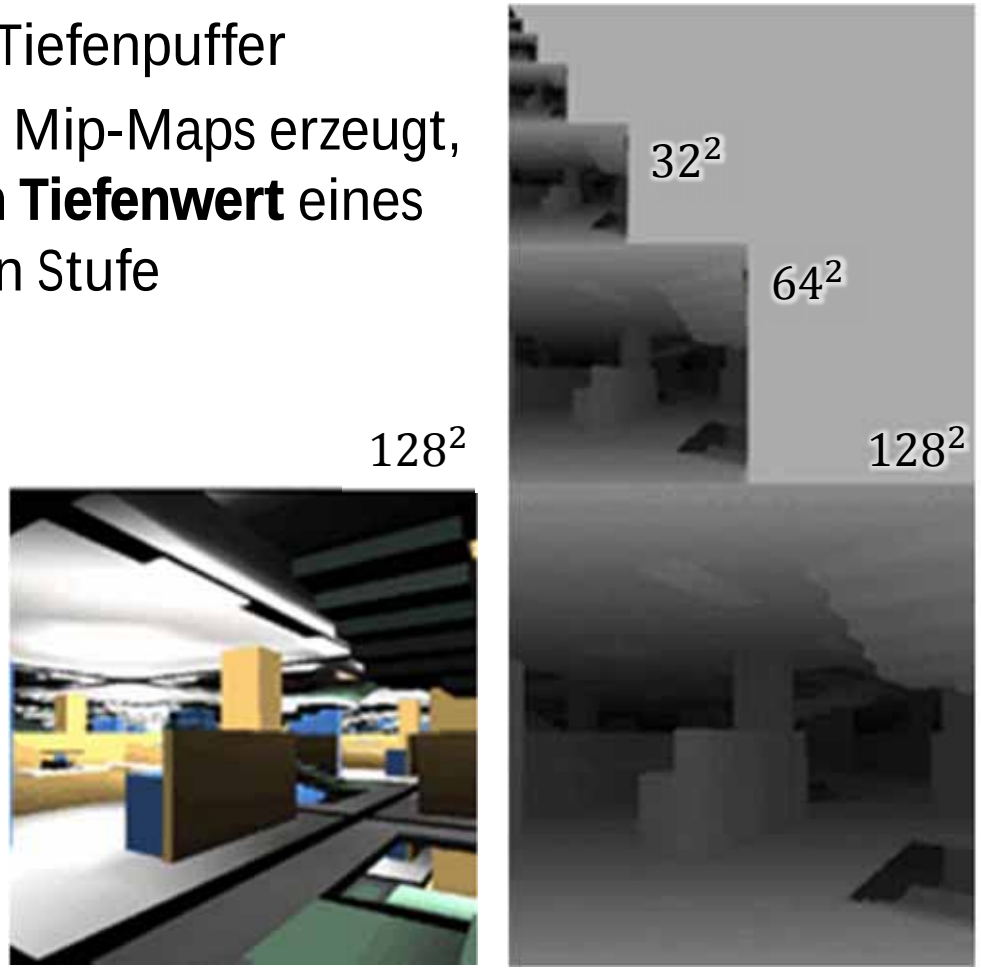
```
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE )
Depth-Only Pass
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE )
glDepthMask( GL_FALSE )

für jede Lichtquelle
  für jedes Objekt (im Einflussbereich der LQ)
    framebuffer += Beleuchtung( Objekt, Licht )
```

- trotzdem fallen Aufgaben mehrfach an:
  - Geometrieverarbeitung und Rasterisierung
  - Texturfilterung (aber nur für sichtbare Flächen)

Hierarchischer Z-Buffer

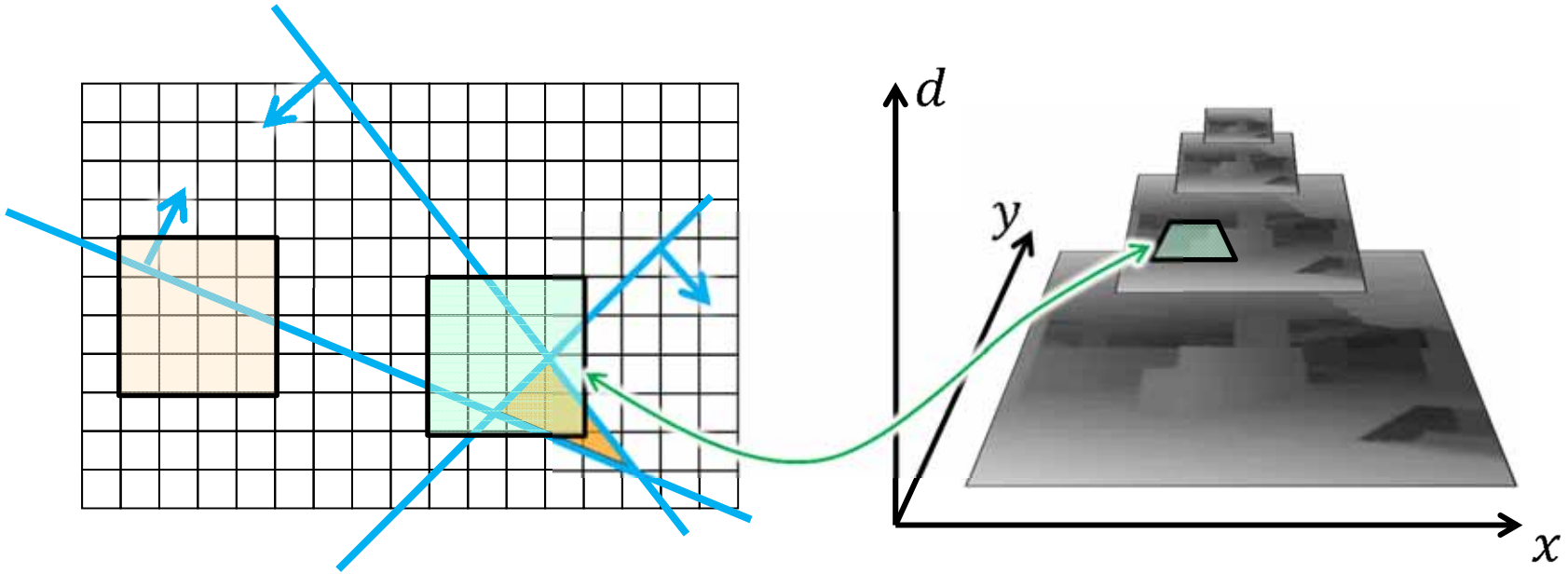
- Ausschließen von verdeckten Flächen für Fragment-Programme findet im Bildraum auf Pixel-Blöcken (4^2 oder 8^2) statt (Funktionalität ist transparent für den Programmierer)
- Idee: HW speichert **hierarchische Tiefeninformation**
  - feinste Stufe = herkömmlicher Tiefenpuffer
  - gröbere Stufen werden ähnlich Mip-Maps erzeugt, speichern aber den **maximalen Tiefenwert** eines 2×2 Blocks der nächstfeineren Stufe



Hierarchical Z-Buffer Visibility,
Greene et al., SIGGRAPH‘93

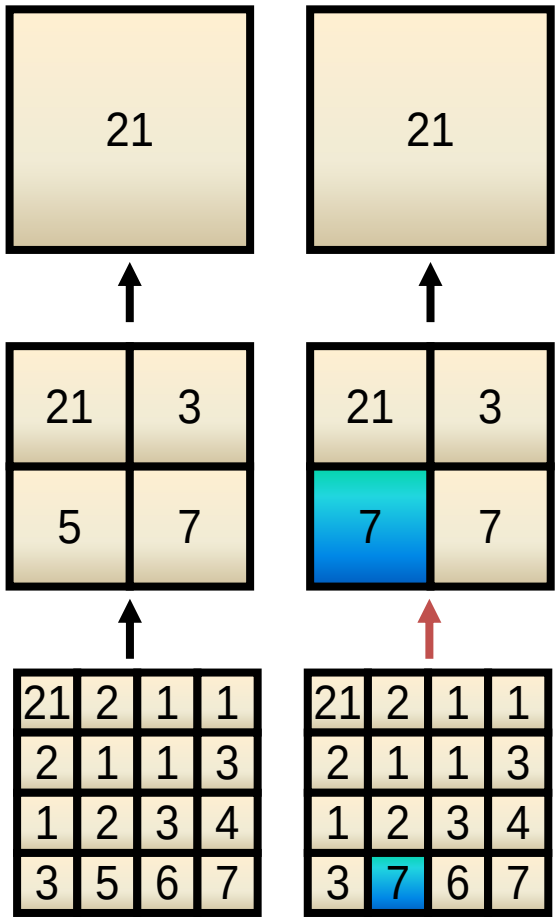
Hierarchischer Z-Buffer

- GPUs verwenden zur Rasterisierung den Kantentest-Ansatz mit Binning
- zuerst werden größere Pixel-Blöcke getestet, ob sie Teile des Dreiecks enthalten, dann (rekursiv) kleinere
- für jeden Block der kleiner-gleich 4^2 bzw. 8^2 Pixel ist wird…
  - … Stufe in der z-Pyramide bestimmt, in der 1 Texel den Block abdeckt
  - der Block kann nicht sichtbar sein, wenn der minimale Tiefenwert des Dreiecks dort größer als die Tiefe in der z-Pyramide ist



Hierarchischer Z-Buffer

- bei Änderungen des Tiefenpuffers muss die Hierarchie durch Propagieren der geänderten Werte konsistent gehalten werden
- implementiert in moderner Grafikhardware
  - z.B. ATI HyperZ-Technology, NVIDIA Early-Z Rejection
  - funktioniert nur, wenn
    - Fragment-Shader keine Tiefenwerte verändern
    - der Tiefentest innerhalb eines Rendering-Durchgangs nicht geändert wird
- idealerweise zeichnet man die Szene trotzdem von vorne nach hinten: der HZB funktioniert auch innerhalb eines Rendering-Durchgangs



Deferred Shading

- bei allen bisherigen sog. „Forward Rendering“-Strategien fallen Aufgaben mehrfach an: Geometrieverarbeitung, Rasterisierung, Texturfilterung
- Deferred Shading (DS) löst diese Probleme (schafft aber neue)
  - DS besteht aus (mindestens) 2 Rendering-Durchgängen
    - **Geometriephase**: Sammeln der **Information im Bildraum** über sichtbare Flächen für die Beleuchtungsberechnung
    - **Beleuchtungsphase**: Beleuchtungsberechnung nur für sichtbare Flächen, unabhängig von der Geometrie der Szene
- Bildraum-Information kann für eine Reihe weiterer Berechnungen genutzt werden, z.B.
  - morphologisches Anti-Aliasing
  - approximatives Ambient Occlusion, indirekte Beleuchtung
  - Kantenfilter für stilistisches oder nicht-photorealistisches Rendering (NPR), ...

Deferred Shading: Geometriephase

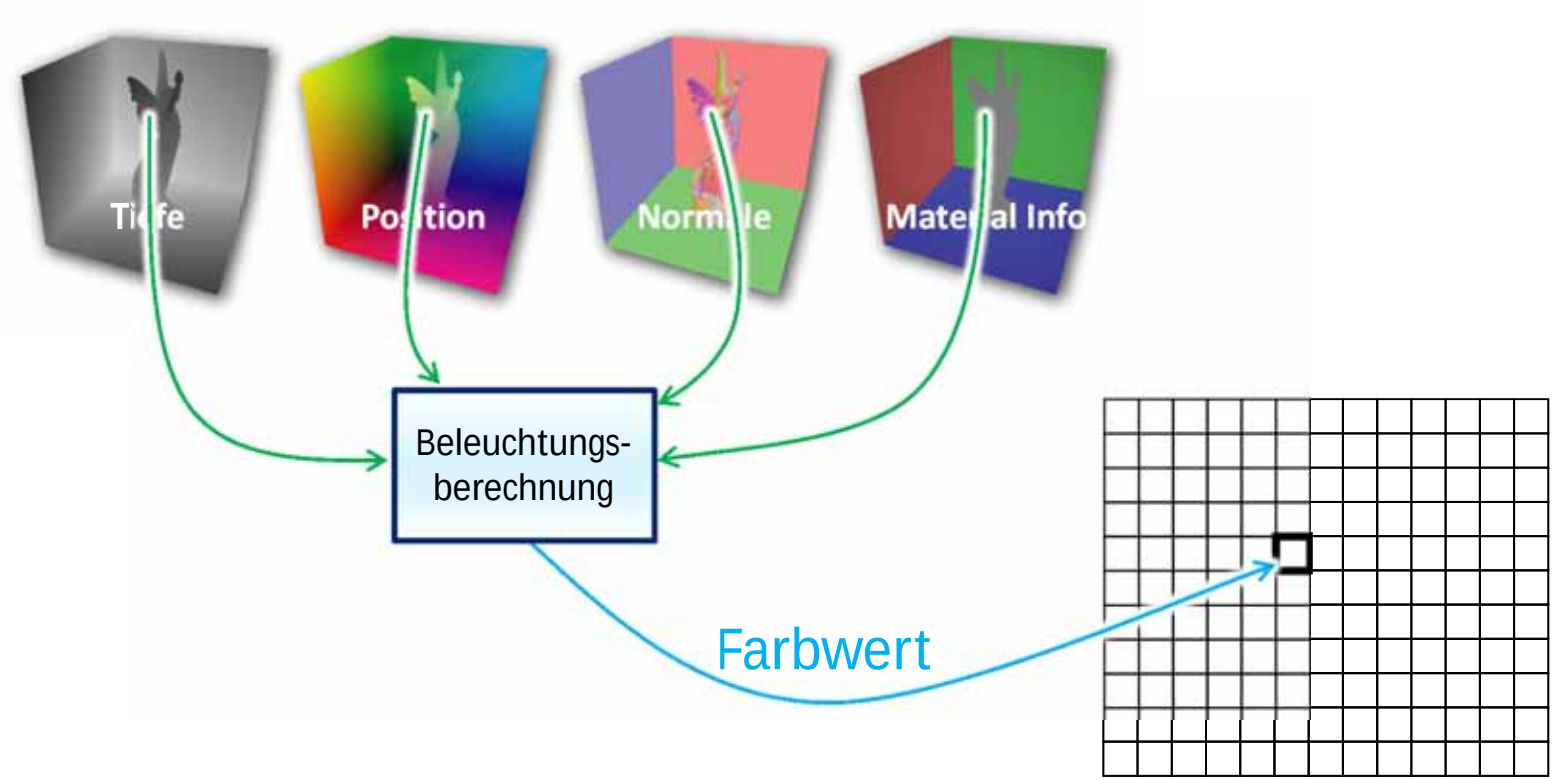
- Füllen des Geometric-Buffer / Geometry-Buffer

```
für jedes Objekt:
  schreibe Oberflächeninformationen in „G-Buffer“
```

  - speichere sämtliche Information, die zur Beleuchtungsberechnung an einem Pixel notwendig ist: Position, Normale, BRDF Parameter, ...
  - dazu werden mehrere Texturen in Rendering-Auflösung verwendet:

Deferred Shading: Beleuchtungsphase

- im G-Buffer ist die Information für die Beleuchtungsberechnung eines Pixels gespeichert → führe ein Fragment-Programm für jeden Pixel aus
- dem Fragment-Programm stehen folgende Daten zur Verfügung
  - „globale Informationen“: Position der Lichtquellen und Kamera, Shadow Maps etc.
  - die Texturen des G-Buffers



Deferred Shading: Beleuchtungsphase

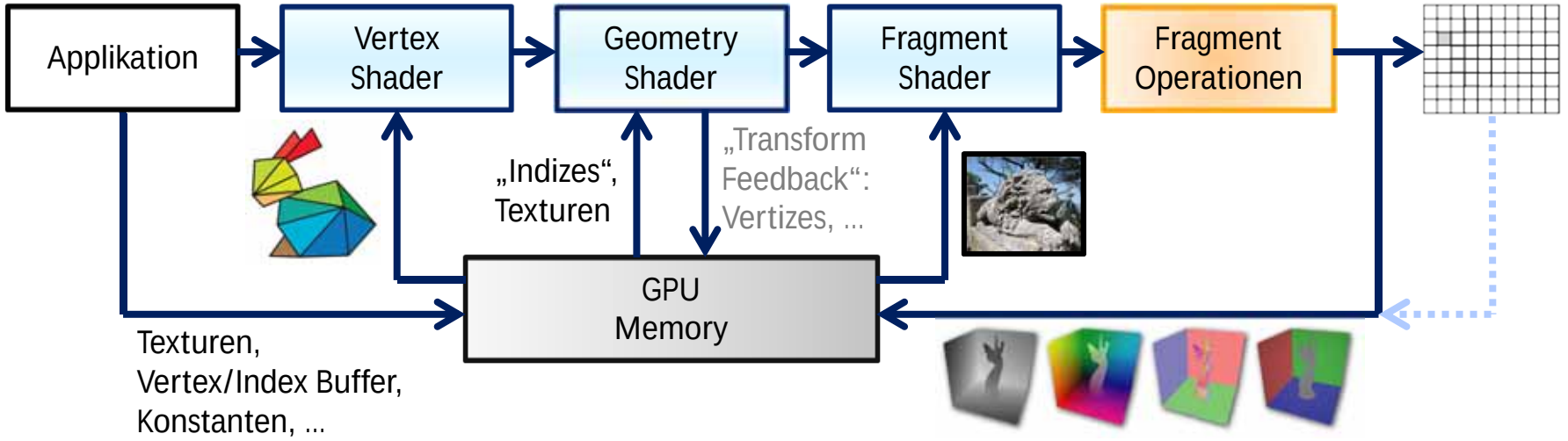
- im G-Buffer ist die Information für die Beleuchtungsberechnung eines Pixels gespeichert → führe ein Fragment-Programm für jeden Pixel aus
- dem Fragment-Programm stehen folgende Daten zur Verfügung
  - „globale Informationen“: Position der Lichtquellen und Kamera, Shadow Maps etc.
  - die Texturen des G-Buffers
- für jede Lichtquelle wird ein bildfüllendes Rechteck gezeichnet: damit wird für jeden Pixel ein Fragment-Programm ausgeführt

```
für jede Lichtquelle:
  für jeden Pixel im Framebuffer
    Pixel += Beleuchtung( G-Buffer, LQ )
```

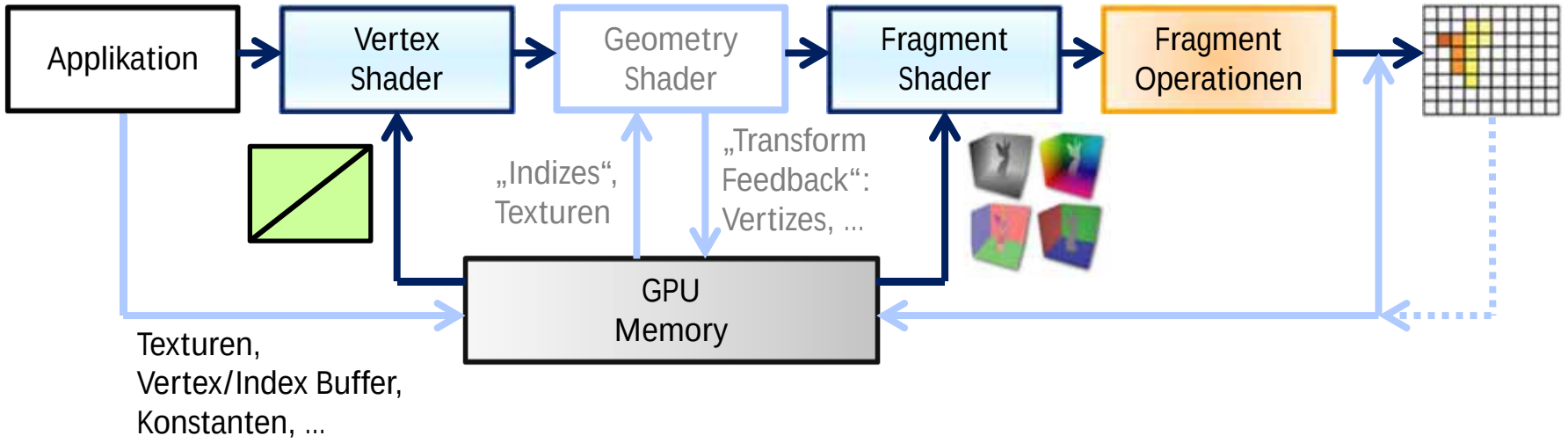
  - Akkumulation der Beiträge der Lichtquellen erfolgt im Framebuffer oder FBOs mittels additivem Blending (`GL_ONE`)

Deferred Shading: Überblick

- Geometriephase: Rendering der Szenengeometrie



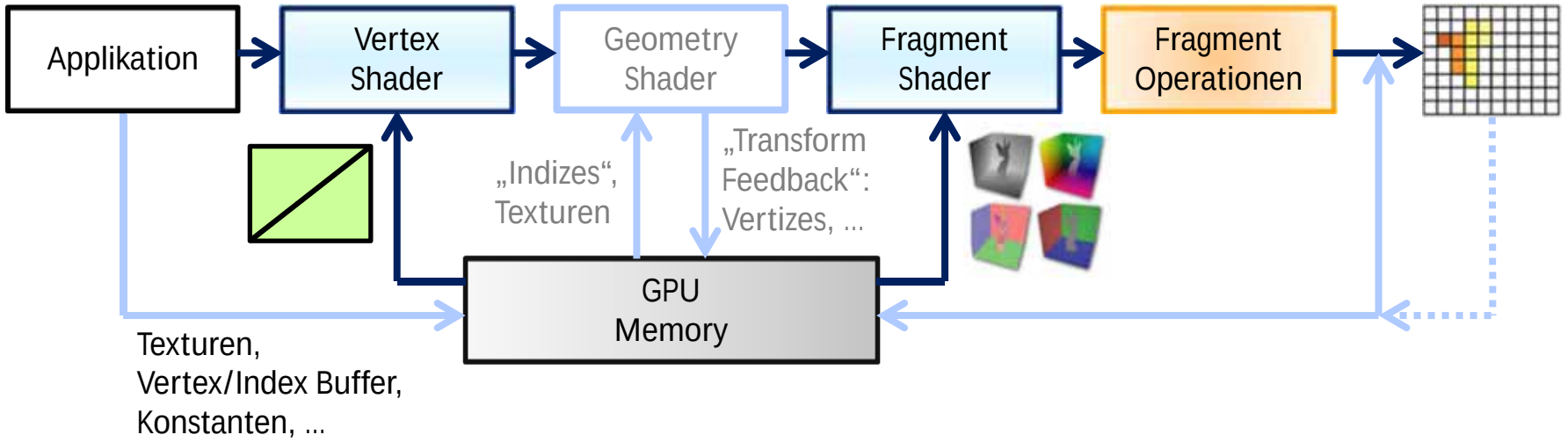
- Beleuchtungsphase: Rendering eines Rechtecks (u.U. mehrfach)



Deferred Shading

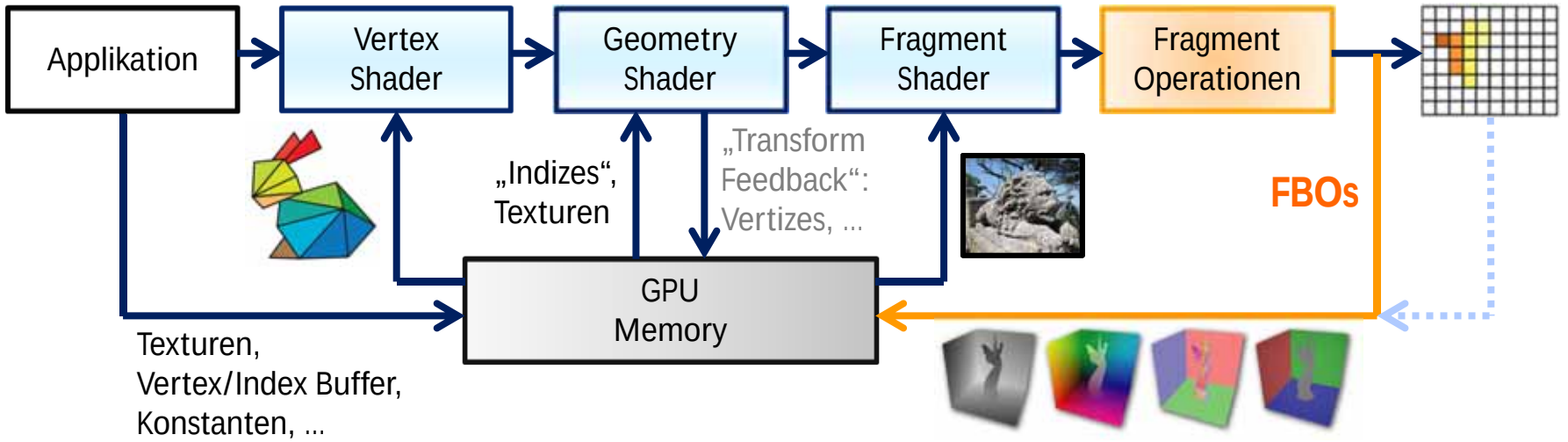
Vorteile

- keine Beleuchtungsberechnungen für verdeckte Flächen
- viele kleine Lichtquellen können effizient behandelt werden (und sind evtl. nicht teurer als eine große/helle Lichtquelle)
- vereinfacht komplexere Rendering-Systeme durch Entkopplung von Geometrie und Beleuchtung
- Idee von Deering et al. (1988), praktikabel seit ca. 2002 (GPUs mit Floating Point) – heute weit verbreitet, z.B. in Videospielen
- wir besprechen natürlich moderne Varianten und aktuelle Techniken



Deferred Shading: praktische Aspekte

- betrachten wir einen „minimalen“ G-Buffer (naiver Ansatz)
  - pro Pixel benötigen wir Position (3 Floats) und Normale (3 Floats)
  - Material: diffuse/spekulare Farbe und Phong-Exponent (7 Floats)
  - bei einer FullHD Auflösung ist der Speicherbedarf damit $1920 \times 1080 \times 13 \times 4$ Byte ≈ 102.8 MB
- zur Erzeugung von G-Buffers verwenden wir **FBOs**
  - leiten die Ausgabe in eigene „Off-Screen“ Buffer
  - Color Buffers: Texturen in die Farbwerte geschrieben werden
  - Renderbuffers: Tiefen- und Stencil-Buffer
- wir betrachten **kurz** die Funktionsweise in OpenGL



Frame Buffer Objects (FBOs)

Schritt 1: Anlegen der Texturen für den eigenen Framebuffer

- erzeuge 4 Texturen im Format `GL_RGBA32F`, d.h. mit je 4 Floats pro Texel

```
// Erzeugen von 4 Texturobjekten
Gluint gbuffer[ 4 ];
glGenTextures( 4, gbuffer );

for ( int i = 0; i < 4; i++ ) {
  glBindTexture( GL_TEXTURE_2D, gbuffer[ i ] );

  // Dummy-Aufruf, damit OpenGL weiß welches Texturformat wir
  // anlegen möchten (alternativ z.B. auch GL_RGBA16F)
  glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA32F,
                width, height, 0, GL_RGBA, GL_FLOAT, NULL );
}
```

Frame Buffer Objects (FBOs)

Schritt 2: Renderbuffer und FBO anlegen

- als nächstes legen wir die OpenGL Objekte für FBOs und RBOs an

```
// jetzt erzeugen wir die Frame Buffer Objects...
glGenFramebuffers( 1, &frameBufferObject );
glBindFramebuffer( GL_FRAMEBUFFER, frameBufferObject );

// ...und verbinden sie mit den Texturen
for ( int i = 0; i < 4; i++ )
  glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + i,
                          GL_TEXTURE_2D, gbuffer[ i ], 0 );
```

Frame Buffer Objects (FBOs)

Schritt 2: Renderbuffer und FBO anlegen

- als nächstes legen wir die OpenGL Objekte für FBOs und RBOs an

```
// jetzt erzeugen wir die Frame Buffer Objects...
glGenFramebuffers( 1, &frameBufferObject );
glBindFramebuffer( GL_FRAMEBUFFER, frameBufferObject );

// ...und verbinden sie mit den Texturen
for ( int i = 0; i < 4; i++ )
  glFramebufferTexture2D( GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0 + i,
                          GL_TEXTURE_2D, gbuffer[ i ], 0 );


// jetzt Erzeugen wir noch das Renderbuffer Objekt
// (letztendlich unseren Tiefenpuffer)
Gluint renderBufferObject;
glGenRenderbuffers( 1, &renderBufferObject );
glBindRenderbuffer( GL_RENDERBUFFER, renderBufferObject );
glRenderbufferStorage( GL_RENDERBUFFER,
                       GL_DEPTH_COMPONENT24, width, height );

// ...und verbinden mit dem aktuell gewählten FBO
glFramebufferRenderbuffer( GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                           GL_RENDERBUFFER, renderBufferObject );
```

Frame Buffer Objects (FBOs)

Schritt 3: Rendering und Verwenden der G-Buffers

- ...analog zu anderen OpenGL Objekten...

```
glGetIntegerv ( GL_DRAW_BUFFER, &curRenderTarget );

glBindFramebuffer( GL_FRAMEBUFFER, frameBufferObject );

const GLenum buffers[4] = {
     GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
     GL_COLOR_ATTACHMENT2, GL_COLOR_ATTACHMENT3 };
glDrawBuffers( 4, buffers );

drawScene(); // hier Zeichnen der Szene...

glDrawBuffers( 1, curRenderTarget );
glBindFramebuffer( GL_FRAMEBUFFER, 0 );

// jetzt können die G-Buffer Texturen verwendet werden, z.B. mit
glBindTexture( GL_TEXTURE_2D, gbuffer[0] ); ...
```

- nicht vergessen: Binding der Fragment-Shader Variablen

```
glBindFragDataLocation( shader, 0, "out_pos" );
glBindFragDataLocation( shader, 1, "out_nrml" );
```

```
out vec4 out_pos, out_nrml;
void main() { ... out_pos = ...; out_nrml = ...; ... }
```

Deferred Shading

Performance Aspekte, Optimierungen

- Optimierungspotenzial bei den **G-Buffers**: Reduktion der Daten
  - keine höhere Genauigkeit als nötig, z.B. müssen weder diffuse Farbe noch Normalen in Floating Point Genauigkeit gespeichert werden
  - Eliminieren von redundanten und leicht zu berechnenden Daten
  - weniger Bandbreite (bei Erzeugung und Lesen der G-Buffer) und (somit) weniger Render-Targets bedeuten höhere Geschwindigkeit

- Optimierung der **Beleuchtungsphase**
  - reduziere Anzahl der Beleuchtungsberechnungen
  - Ausnutzen von Materialeigenschaften

Optimierungen: Geometriephase

- Speicherung der Normale $\boldsymbol{n} = (n_x, n_y, n_z)^T$ mit $|\boldsymbol{n}| = 1$
  - Darstellung über sphärische Koordinaten (2 Winkel): Umwandlung erfordert (vergleichsweise teure) trigonometrische Funktionen
  - bei der Beleuchtungsberechnung in Kamerakoordinaten gilt:
    - sichtbare Flächen zeigen natürlich zum Betrachter hin
    - die z-Komponente der Normale ist also immer positiv

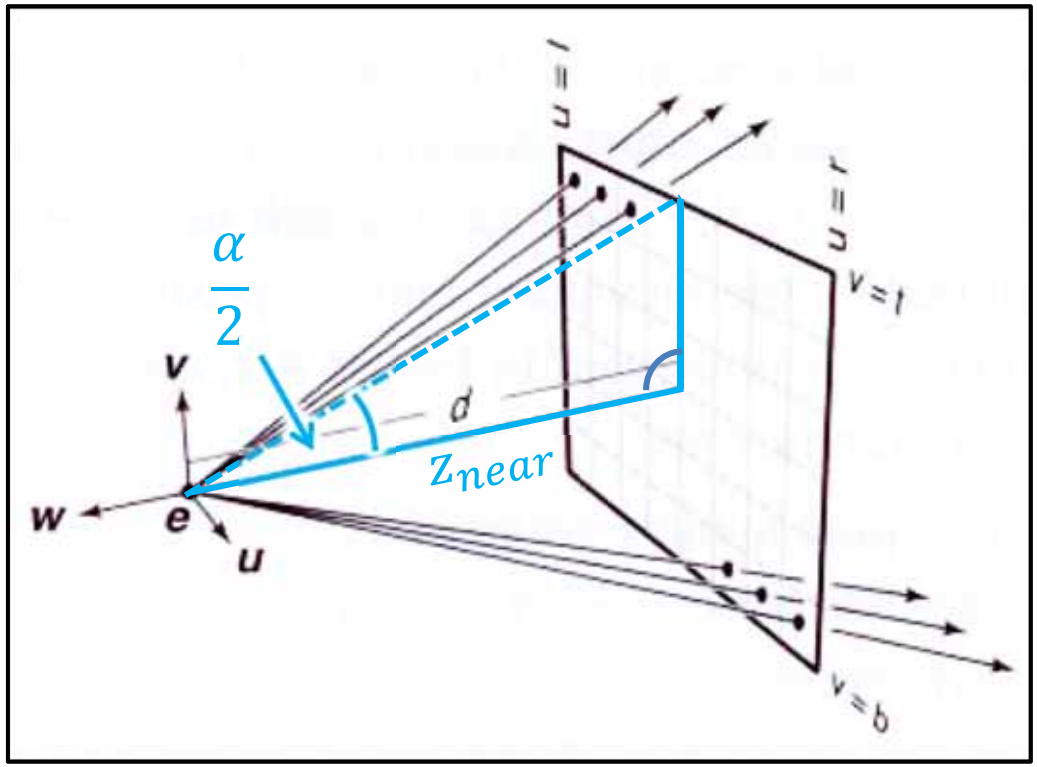
      $\Rightarrow n_z = \left(1 - {n_x}^2 - {n_y}^2\right)^{-1/2}$

- weitere Optimierung: n_x und n_y erfordern keine FP-Genauigkeit
  - es ist möglich Werte zu packen, z.B. 4 Byte-Werte in einen 32 Bit Float zu speichern (oder 1 Float in einem 32-Bit-RGBA-Quadrupel)
  - Berechnung entweder durch Multiplikation und **fract**
    http://smt565.blogspot.com/2011/04/bit-packing-depth-and-normals.html
  - oder mittels GLSL >3.3 Befehlen: **float intBitsToFloat(uint x);**
    siehe: http://www.opengl.org/sdk/docs/manglsl/xhtml/intBitsToFloat.xml

Optimierungen: Geometriephase

Berechnung der Position aus Tiefe und Kamera

- die Position eines Pixels in Weltkoordinaten ist eindeutig bestimmt durch
  - die Kameraabbildung (Öffnungswinkel α, Aspect Ratio $^{r}/_{t}$)
  - seinen Tiefenwert (= Entfernung zur Kamera) und
  - die Pixel-Koordinate im G-Buffer
  - es besteht also keine Notwendigkeit die Position zu speichern



Optimierungen: Beleuchtungsphase

Performance Aspekte

- Kosten für additives Blending (Akkumulation der Lichtquellen-Beiträge)
  - Beiträge einer Lichtquelle können klein sein, u.U. ist Blending in einem 8-Bit-RGB(A) Framebuffer nicht ausreichend
  - Blending in einem Floating Point Framebuffer meist nicht vermeidbar → etwas teurer und erhöhte Speicherbandbreite, aber kein allzu großes Problem

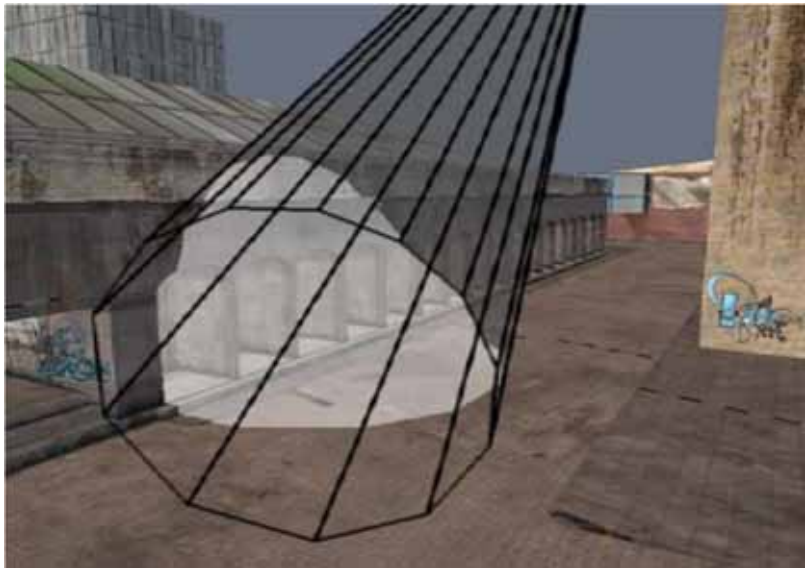
- Kosten der Beleuchtungsberechnung
  - hängen vom Beleuchtungsmodell/BRDF ab (Phong, Cook-Torrance, ...)
  - sind **proportional zur Anzahl der beleuchteten Pixel**

- bisher: „globale“ Lichtquellen, die gesamte Szene/Bild beeinflussen
  - durch Zeichnen eines bildschirmfüllenden Rechtecks (das Ziel war für jeden Pixel ein Fragment-Programm auszuführen)

Optimierungen: Beleuchtungsphase

Idee: Lokale Lichtquellen

- oft haben Lichtquellen nur Einfluss auf einen Teil des Bildes
  - z.B. Einfluss einer Punktlichtquelle im Abstand r ist $\propto 1/r^2$,
    ab einer bestimmten Entfernung ist der Beitrag vernachlässigbar
- der Einflussbereich lässt sich mit Hüllkörpern beschreiben
  - Punktlichtquelle → Kugel
  - Spot Light → Kegel
  - direktionale Lichtquelle → Quader
- Idee: ein Fragment-Programm soll nur im Einflussbereich ausgeführt
  - zeichne dazu die Hüllkörper
    (mit Kameraabb. und Tiefentest)

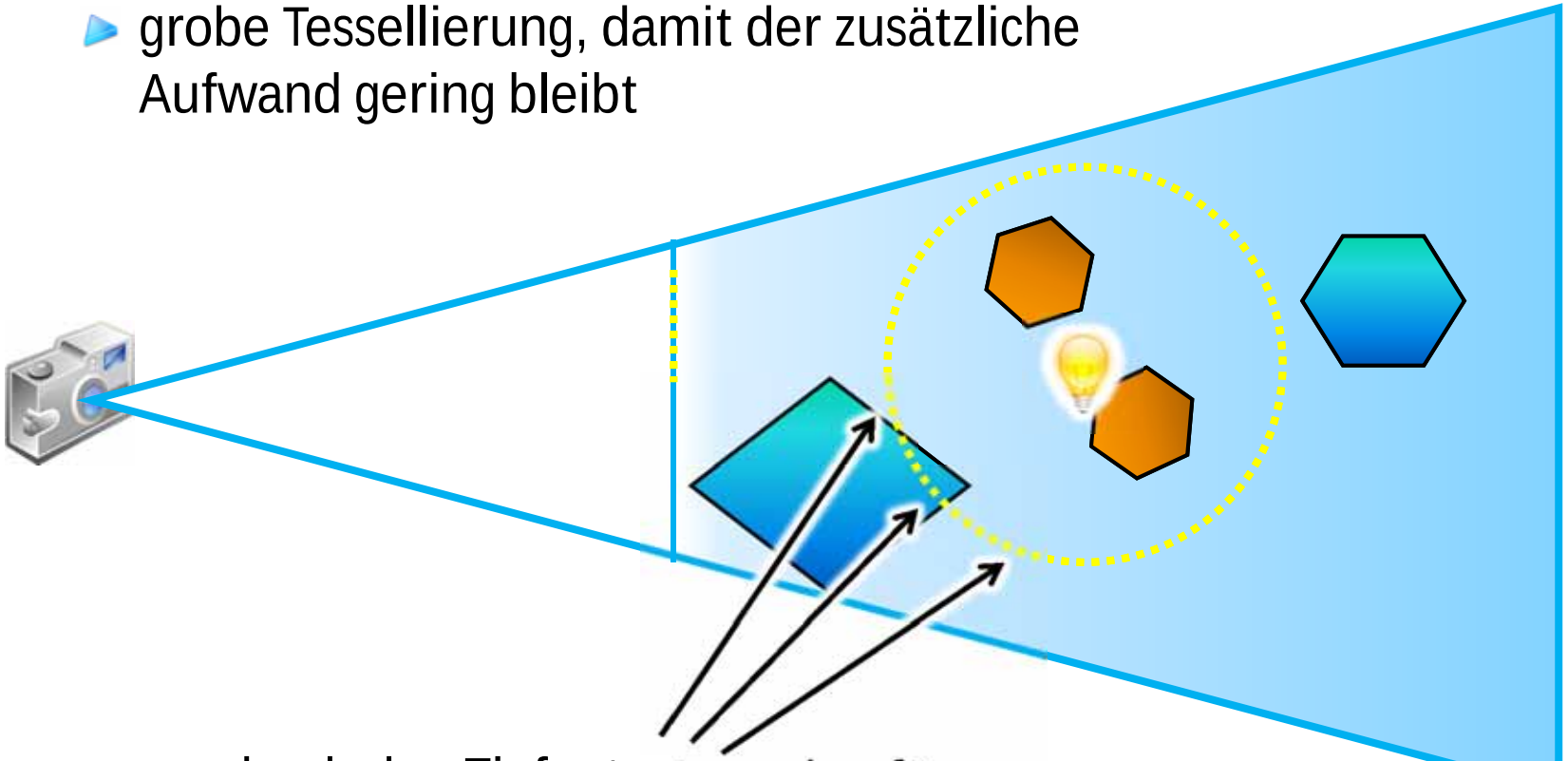


Shawn Hargreaves, GDC2004

Optimierung durch Hüllkörper

Beispiel: Punktlichtquelle

- zeichne eine Kugel anstatt eines Rechtecks über das ganze Bild
  - grobe Tessellierung, damit der zusätzliche Aufwand gering bleibt

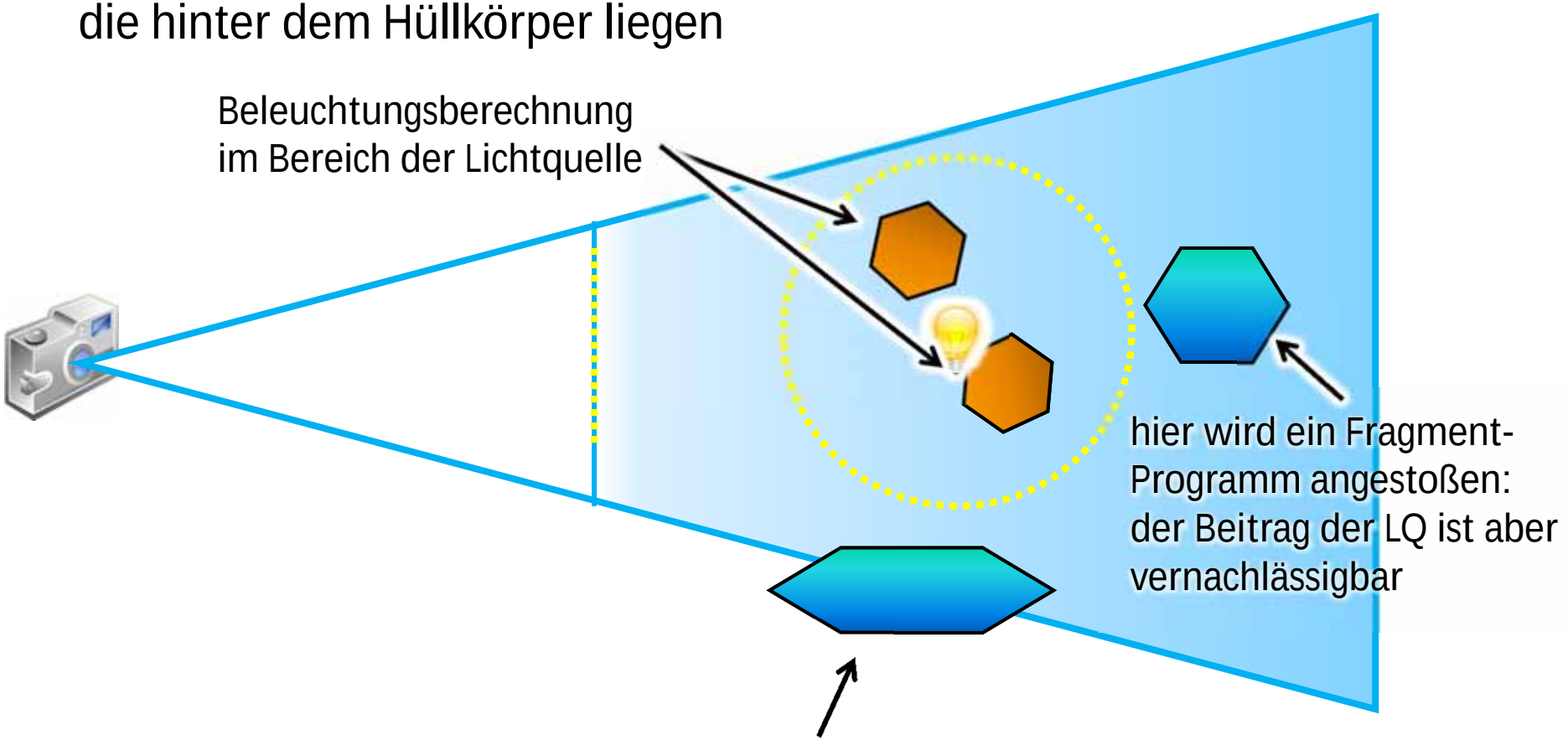


durch den Tiefentest werden für verdeckte Teile der Hüllkörper keine Beleuchtungsberechnungen durchgeführt

Optimierung durch Hüllkörper

Beispiel: Punktlichtquelle

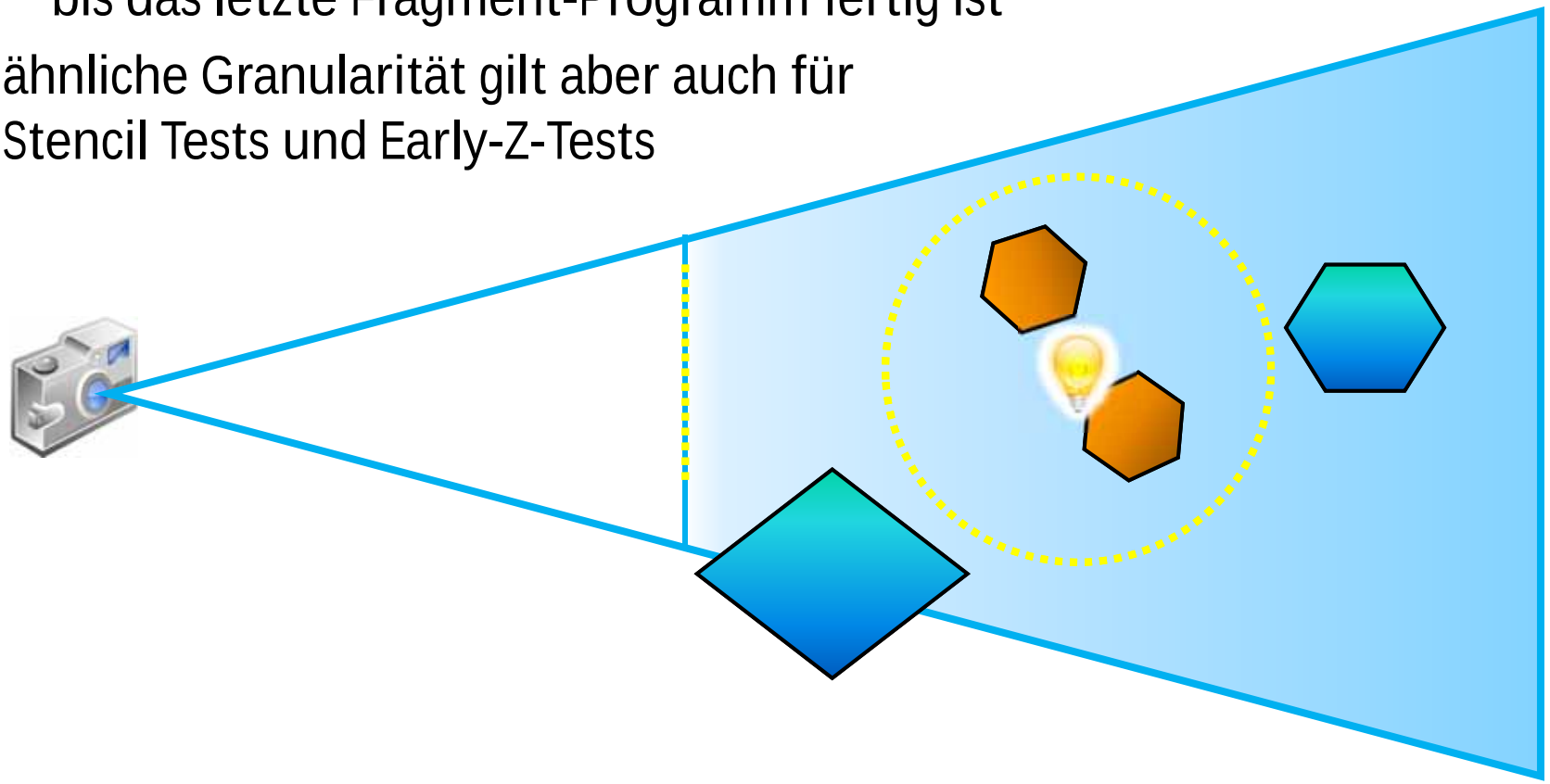
- es gibt noch weiteres Optimierungspotenzial bei Szenenteilen, die hinter dem Hüllkörper liegen



Optimierung durch Hüllkörper und „Discard“

Discard in Fragment Programmen

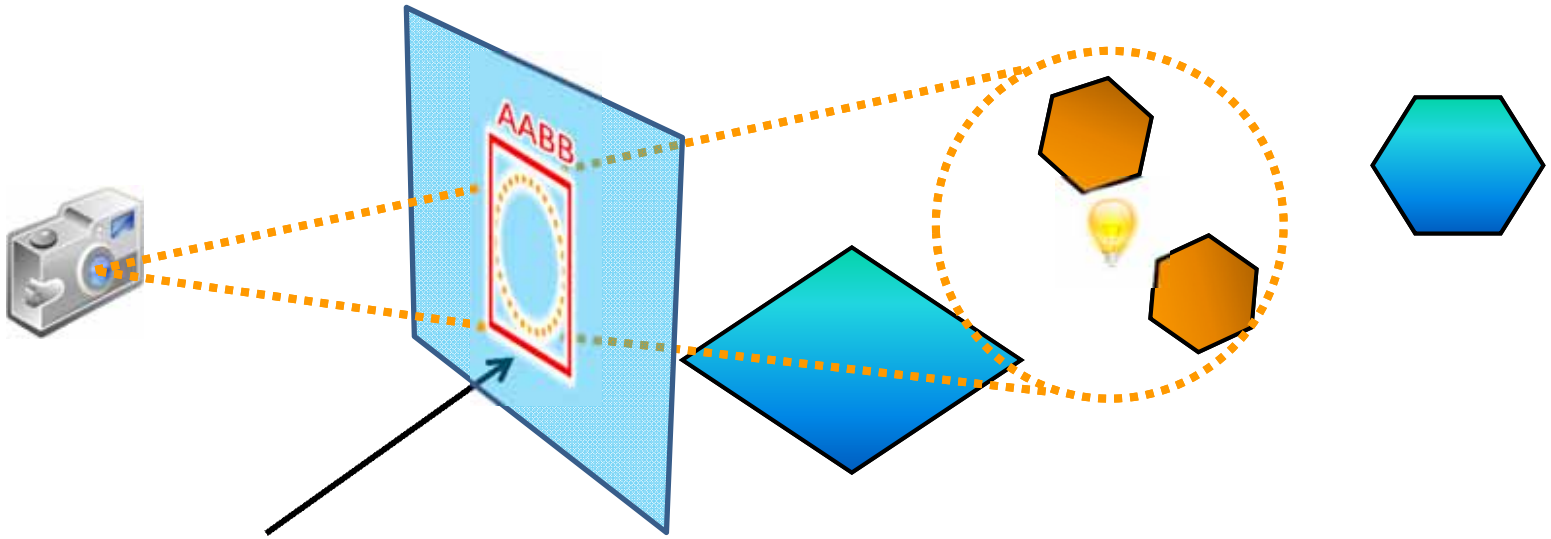
- `discard`-Befehl (nur in Fragment-Programmen) beendet die Ausführung ohne ein Fragment zu Erzeugen: verwerfe bei falschem Tiefenwert
- Achtung: FP werden immer in Blöcken von 4^2, 8^2, … Pixel ausgeführt
  - Threads starten zum gleichen Zeitpunkt und sind alle blockiert, bis das letzte Fragment-Programm fertig ist
- ähnliche Granularität gilt aber auch für Stencil Tests und Early-Z-Tests



Optimierung durch 2D-Hüllkörper

Beispiel: Punktlichtquelle

- oft werden auch 2D-Hüllkörper **im Bildraum** verwendet, z.B. AABBs
- Vorteil: geringer Aufwand in der Geometrieverarbeitung, einfach zu berechnen und nur wenig unnötig rasterisierte Fläche

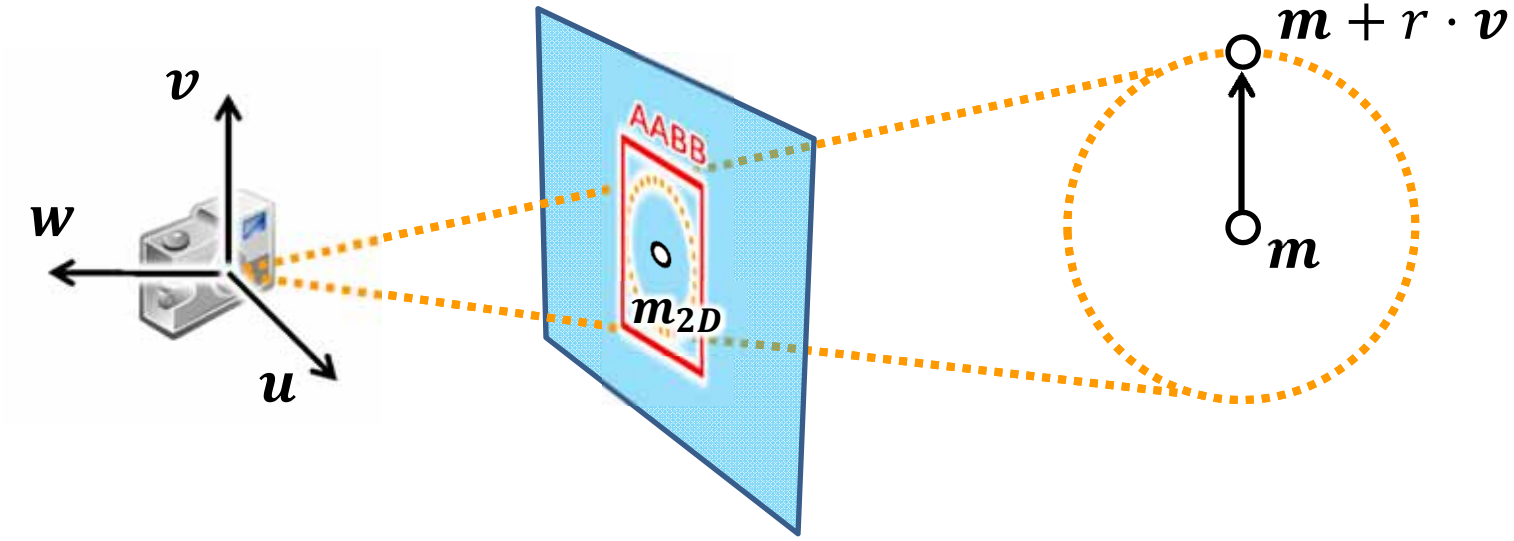


Beleuchtungsberechnung im Einflussbereich der LQ wird durch Zeichnen eines Rechtecks angestoßen (Zeichnen ohne Tiefentest, additives Blending)

Optimierung durch 2D-Hüllkörper

Beispiel: Berechnung einer AABB einer Kugel

- Idee: projiziere den Kugelmittelpunkt und Punkt auf der Silhouette auf Bildebene (folgende Berechnung betrachtet *nicht exakt* die Silhouette)
- allg. mit dem Kamerakoordinatensystem $\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{w}$ kann man Flächen senkrecht/parallel zur Blickrichtung (in Weltkoord.) aufspannen
- geg. Mittelpkt. $\boldsymbol{m}$ in homogenen Koord., View-Projection-Matrix $\mathbf{M_{VP}}$
  - Projektion $\boldsymbol{m_{2D}} = \mathbf{M_{VP}} \cdot \boldsymbol{m}$
  - Punkt auf dem Rand (Radius r): $\boldsymbol{p} = \boldsymbol{m} + r \cdot \boldsymbol{v}$ und $\boldsymbol{p_{2D}} = \mathbf{M_{VP}} \cdot \boldsymbol{p}$
  - Kantenlänge der AABB (zentriert um $\boldsymbol{m_{2D}}$): $2|\boldsymbol{p_{2D}} - \boldsymbol{m_{2D}}|$
    - **Dehomogenisieren nicht vergessen!**



Beleuchtungsphase Optimierungen

Optimierung der Bandbreite: Light Pre-Pass Rendering

- Idee: Ausklammern von Faktoren im Beleuchtungsmodell, die unabhängig von den Lichtquellen sind, z.B. diffuse Farbe
  - Beleuchtungsberechnung mit mehreren LQ: $I = \sum_i k_d \cdot I_{L_i} \cdot (\mathbf{N} \cdot \mathbf{L})$
  - geänderte Berechnungsreihenfolge ändern: $I = k_d \cdot \sum_i \left(I_{L_i} \cdot (\mathbf{N} \cdot \mathbf{L}) \right)$
- Vorgehen: akkumuliere die Lichtquellenbeiträge und greife nur einmal am Ende auf die Materialparameter zu → Einsparung von Bandbreite (radiometrische Interpretation: wir speichern Irradiance)
- Trennung bei spekularen Termen ist bsp. bei den Phong-Exponenten nicht möglich: $k_s \cdot \sum_i \left(I_{L_i} \cdot (\mathbf{R} \cdot \mathbf{V})^n \right)$

Beleuchtungsphase Optimierungen

Light Pre-Pass Rendering

- Rendering der Beleuchtungsphase in weitere Texturen, in denen diffus (und evtl. eben doch spekular) reflektiertes Licht gespeichert wird:

```
für jede Lichtquelle:
  renderTarget[ 0 ] += diffuse( P, N, L )
  renderTarget[ 1 ] += specular( P, N, n, L )
```

- ein abschließender Rendering-Durchgang berechnet die endgültige Farbe

```
framebuffer = renderTarget[ 0 ] * G-Buffer.Kd +
              renderTarget[ 1 ] * G-Buffer.Ks;
```

Schatten

- mit Deferred Shading kann eine große Zahl von Lichtquellen verwendet werden – das Erzeugen der Shadow Maps ist nach wie vor aufwändig
  - Shadow Volumes werden praktisch nicht mit Deferred Shading verwendet: das Verfahren würde funktionieren, aber die Grundidee „Entkoppeln von Geometrie und Beleuchtung“ gilt dann nicht mehr
- oft werfen nur die hellsten/wichtigsten Lichtquellen Schatten
  - dann wird das Resultat des Schattentests im G-Buffer gespeichert
  - Beispiel: CryEngine 2

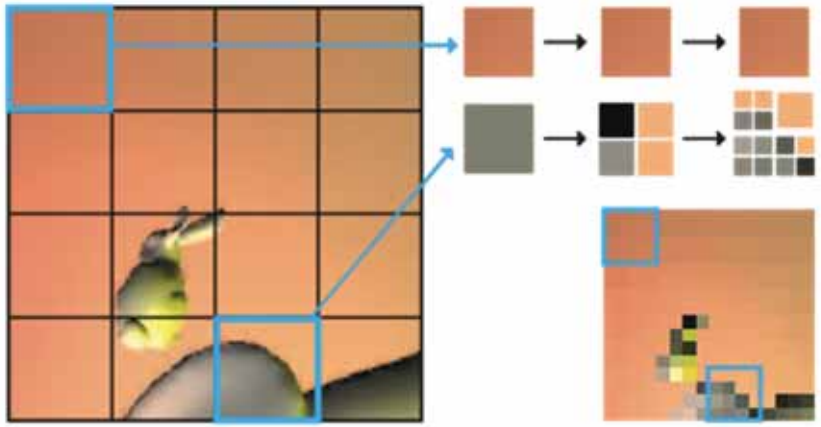
Verschiedene Materialien

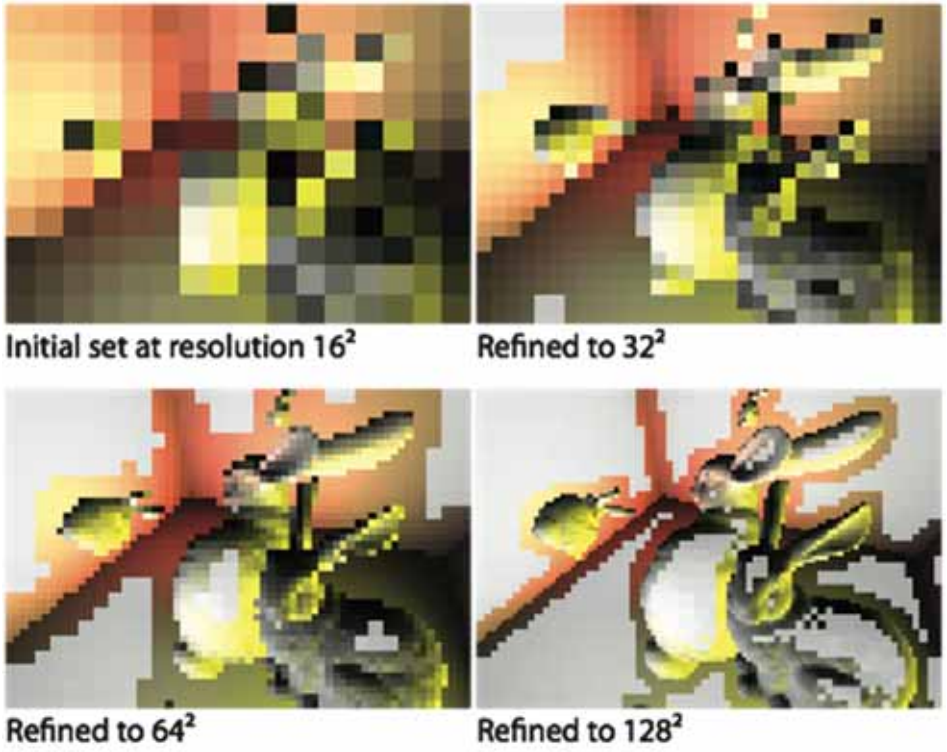
- auch Deferred Shading „leidet“ unter der Vielfalt von Lichtquellen und Materialkombinationen
  - Ausführen eines Fragment-Programms pro Lichtquelle, nicht pro Material
  - d.h. die Materialparameter sollten im G-Buffer gespeichert sein
  - Widerspruch zum Ziel die Menge der Daten pro Pixel gering zu halten

- mögliche Lösungen
  - Multi-Pass: zeichne jeden Hüllkörper einer Lichtquelle einmal pro Materialtyp – die Daten im G-Buffer werden dann entsprechend interpretiert → zu viele Rendering-Durchgänge
  - besser: speichere den Materialtyp (ID) im G-Buffer und lese die Materialparameter aus einer Tabelle aus (Verzeigung im Fragment-Programm)

Ausblick: Adaptive Verfeinerung (Nichols and Wyman)

- erzeuge **„Min-Max-Mipmap“** des Tiefenpuffers zur Detektion von Diskontinuitäten
- verwende Auflösungshierarchie für das Shading
  - berechne Beiträge einer Lichtquelle in grober Auflösung
  - … aber nur, wenn keine großen Diskontinuitäten in diesem Bereich
- am Ende: Interpolation und Compositing der Bilder





Ausblick: Tiled and Clustered Shading

Grundidee (Olsson et al., HPG 2012)

- erzeuge eine räumliche Hierarchie von Punktlichtquellen
- unterteile Sichtpyramide im Bildraum (2D-Tiles) und in der Tiefe
- bestimme Cluster von Lichtquellen, die für ein Tile benötigt werden

http://www.cse.chalmers.se/~olaolss/